C12

```
  1
  2          /*
  3    ** ======================================================
  4    **
  5    ** Copyright 1996, 1997 EMC Corporation
  6    **
  7    ** ======================================================
  8    **
  9    ** DDMSvc_init.c
 10    **
 11    ** ======================================================
 12    **
 13    ** Mission Statement:
 14    **
 15    ** ======================================================
 16    **
 17    ** Primary Data Acted On:
 18    **
 19    ** ======================================================
 20    **
 21    ** Compile-Time Options:
 22    **
 23    **      USE_SUNRPC - Compile source with sunrpc
 24    **                   support.  If
 25    **                   not set, assume DCE support.
 26    **
 27    ** ======================================================
 28    **
 29    ** Basic idea here:
 30    **
 31    ** ======================================================
 32    **
 33    ** ======================================================
 34    **
 35    ** The following provides an RCS id in the binary that can be located
 36    ** with the what(1) utility. The intent is to keep this short.
 37    **
 38    */
 39   #if !defined(lint)
 40   static char    RCS_Id [] = '@(#)$WCSfile: EDMDsrc.c.v $ "
 41                              "$Revision: 1.23 $ "
 42                              "$Date: 1997/02/06 20:49:15 $" ;
 43   #endif
 44
 45   /* #define _POSIX_SOURCE    unable to compile with this define set */
 46   /* #define _XOPEN_SOURCE    unable to compile with this define set */
 47
 48   #include <sys/types.h>
 49   #include <sys/uio.h>
 50   #include <sys/utsname.h>
 51   #include <sys/socket.h>
 52   #include <netinet/in.h>
 53   #include <arpa/inet.h>
 54   #include <netdb.h>
 55
 56   #include <csl/c_portable.h>
 57   #include <csl/epl_xopen.h>
 58   #include <csl/enlinout.h>
 59
 60   #include <string.h>
 61   #include <stdlib.h>
 62   #include <pthread.h>
 63
 64   // Rogue Wave includes
 65   #include <rw/collect.h>
 66   #include <rw/cstring.h>
 67   #include <rw/tvslist.h>
 68   #include <rw/tvstream.h>
 69   #include <rw/bintree.h>
```

```
 61   #include <sec/secomm.h>
 62   #include <edmlink/edmlink_api.h>
 63
 64   #ifdef __cplusplus
 65   extern "C" {
 66   #endif
 67
 68   #include <restore/dispatch_daemon.h>
 69   #include <restore/dispatch_protocol.h>
 70   #include <restore/sec_dispatch.h>
 71   #include <restore/dispatch_Protocol_Service.h>
 72   #include <restore/sec_Dispatch_Protocol_Service.h>
 73   #include <restore/dispatch_protocol_client.h>
 74   #include <dpservice.h>
 75
 76   #ifdef __cplusplus
 77   }
 78   #endif
 79
 80   #include <logging/logging.h>
 81   #include <EDMDispatchLog.h>
 82   #include <EDMDHandle.h>
 83   #include <EDMDHandleMgrApi.h>
 84   #include <EDMSession.h>
 85   #include <EDMCc.h>
 86   #include <EDMUtils.h>
 87   #include <EDMD_ddp.h>
 88   #include <EDMDDoc_restsrc.h>
 89
 90   pthread_cond_t  cscbortRdy_cv   = PTHREAD_COND_INITIALIZER;
 91   pthread_mutex_t cscbortRdy_mutex = PTHREAD_MUTEX_INITIALIZER;
 92   pthread_mutex_t g_servicemtx;
 93
 94   static boolean12 print_error = TRUE;
 95
 96   /* Prototypes */
 97   int edmrst_send_chdl_to_private_svc(int);
 98   int edmrst_handle_t cscPortRdy_mutex;
 99   int edmrst_create_ddp_client_binding_handle_t ** , EDMSession * );
100
101   int rpc_binding_handle_t ** , EDMSession * );
102
103   int edmrst_send_uid_to_private_svc(int, EDMSession );
104
105   static rpc_if_handle_t Dispatchdaemon_ifspec;
106
107   /* Dispatch Protocol ifspec */
108   static rpc_if_handle_t Dispatchdaemon_ifspec;
109   BlinkHandlePtr_t  BlinkHandle;
110
111   /***********************************************************
112    *
113    * Routine:      LockSvcMutex
114    *
115    * Inputs:       None
116    *
117    * Outputs:      None
118    *
119    * Return Codes:
120    *               None
121    *
122    * Purpose:  Lock the mutex for the service execution
123    *
124    ***********************************************************/
125
126   static void
127   LockSvcMutex()
128   {
```

```
123 1 {
124 1    static boolean_ty first = TRUE;
126 1    if (first == TRUE)
127 2    {
128 2        first = FALSE;
129 2        pthread_mutex_init(&G_serviceMtx, NULL);
130 1    }
132 1    pthread_mutex_lock(&G_serviceMtx);
133 1 }
```

```
135 /*****************************************************************
136 **
137 ** Routine:   UnlockSvcMutex
138 **
139 ** Inputs:    None
140 **
141 ** Outputs:   None
142 **
143 ** Return Codes:
144 **    None
145 **
146 ** Purpose:   Unlock the mutex for service execution
147 **
148 **
149 */
151 static void
152 UnlockSvcMutex()
153 {
154 1    pthread_mutex_unlock(&G_serviceMtx);
155 1 }
```

```
157      void *
158      DDRSrvc_init(void *pSessObj)
159
160  1   {
161  1       int                      lcr;            /* Local Return Code */
162  1       int                      fdj;
163  1       int                      fdj;
164  1       int                      status;
165  1       EPC_binding_handle_t     bh=NULL;
166  1       EDMSession_id *          p_so;
167  1       DC_client_session_id     sid;
168  1       ElinkShellObjPtr_ty      ShellHandle;
169  1       unsigned char            *svc_rpc_handle;   /* X-Service RPC handle */
170  1       ElinkTargetObjPtr_ty     TargetObjPtr;      /* Target object; all functions */
171  1       ElinkUserIdObjPtr_ty     UserIdObjPtr;      /* UserId object copy & shell */
172  1       CmdObjPtr                CmdObjPtr;         /* Shell command shell only */
173  1       unsigned long            options = 0;       /* For ElinkNewCreLaunchObj */
179  1
180  1           LockSvcMutex();
181  1           pthread_mutex_lock( &cscPortRdy_mutex );
182  1
183  1       // Check to see that the EDMLINK handle didn't get trashed
184  1
185  1       }
186  1           UnlockSvcMutex();
187  1           pthread_mutex_unlock( &cscPortRdy_mutex );
188  2           system we want to talk to.
189  2       }
190  1       // (ElinkHandle == NULL)
192  1
193  1       // Cast the input argument to its object type.
194  1
195  1       p_so = (EDMSession *)pSessObj;
197  1
198  1       // Construct EDM-Link target object so that EDM-Link will know what
199  1       // system we want to talk to.
200  1
202  1       TargetObjPtr = ElinkNewTargetObj( ElinkHandle,
203  1                                         "localhost" );
205  1
206  1       // EDM-Link should have called our callback DOMFileAckCallback which
207  1       // should have loaded DOMHandle->ErrorBlock, so all we have to do
208  1       // now, is return.
210  1
211  1       if ( NULL == TargetObjPtr )
212  1       {
213  2
214  1           p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
215  1           UnlockSvcMutex();
216  1           pthread_mutex_unlock( &cscPortRdy_mutex );
                 pthread_exit( NULL );
             }
```

```
218  1       // target.
219  1
220  1       // Construct EDM-Link user object.
221  1       //   We are going to want to run as root on the
222  1       // We know that we will be starting via the EDM-Link daemon and we can
223  1       // always start using the root id. Also, this will be a service, it needs
224  1       // to run as root and will have some intelligence in protecting itself in
225  1       // that there a limited what can be done and the caller of the
226  1       // will control what can be done and the caller of the API
227  2
228  1       UserIdObjPtr = ElinkNewUserIdObj( ElinkHandle,
229  1                                         TargetObjPtr,
230  1                                         NULL,
231  1                                         NULL );
237  1
238  1       // EDM-Link should have called our callback DOMFileAckCallback which
239  2       // should have loaded DOMHandle->ErrorBlock, so all we have to do
240  1       if ( NULL == UserIdObjPtr )
242  1
243  1           p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
244  1           UnlockSvcMutex();
245  1           pthread_mutex_unlock( &cscPortRdy_mutex );
246  1           pthread_exit( NULL );
247  2       }
249  1       // Utilize the EDM-Link service launcher to physically startup the
250  1       // domain private service, a private service can
251  1       // be found in /usr/epoch/service and have a suffix of pd    The domain
253  1       // private service is /usr/epoch/service/domainpd.
254  1
256  1       if ( isDebugOn() )
257  1           options |= ELINK_SERVICE_DEBUG;
258  1                               /* if we are debug, so will be R.Eng */
259  1
260  1       CmdObjPtr = ElinkNewServiceLaunchObj( ElinkHandle,
261  1                                             TargetObjPtr,
262  1                                             UserIdObjPtr,
263  1                                             "edmrestoreeng",
264  1                                             options );
265  1                               /* Domain private service */
266  2
267  1       // EDM-Link should have called our callback DOMFileAckCallback which
268  2       // should have loaded DOMHandle->ErrorBlock, so all we have to do
269  2       // now, is return.
270  2
271  1       if ( NULL == CmdObjPtr )
272  1       {
273  1           (void) ElinkDestroyObj( ElinkHandle, TargetObjPtr );
                 (void) ElinkDestroyObj( ElinkHandle, UserIdObjPtr );
                 p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
                 UnlockSvcMutex();
                 pthread_mutex_unlock( &cscPortRdy_mutex );
                 pthread_exit( NULL );
             }
```

```cpp
275  1
276  1      // Fire up Private Service via EDM-ICM API ElinkPrivateSvc.  This
277  1      // physically starts the private service running.
278  1
279  1      lrc = ElinkPrivateSvc ( ElinkHandle,
280  1                              TargetObjPtr,
281  1                              UserIdObjPtr,
282  1                              CmdObjPtr,
283  1                              &fd1,
284  1                              &fd2,
285  1                              &ShellHandle );
286  1
287  1      if ( -1 == lrc )
288  1      {
289  1
290  1          p_so -> setStatus(DD_SERVICE_FAILURE,
291  2              __FILE__, __LINE__, DDP_PRIVATE_SVC_FAILURE,
292  2              0,"ElinkPrivateSvc() failure");
293  2          pthread_mutex_unlock( &cscPortRdy_mutex);
294  2          pthread_exit( NULL );
295  1      }
296  1
297  1
298  1      // Extract the csc handle from the shell object. This handle
299  1      // is the restore service (restore API) per csc handle.
300  1
301  1      (void) ElinkDestroyObj( ElinkHandle,  TargetObjPtr );
302  1      (void) ElinkDestroyObj( ElinkHandle,  UserIdObjPtr );
303  1      (void) ElinkDestroyObj( ElinkHandle,  CmdObjPtr );
304  1
305  1      svc_rpc_h = (unsigned char*) calloc(1,CONNECT_HANDLE_SIZE);
306  1      if ( svc_rpc_h == NULL )
307  1      {
308  1
309  1          EDMDispatch_logent(
310  2              __FILE__, __LINE__, LOG_ERR, DDP_NO_MEMORY,
311  2              0,"calloc() failure");
312  2          pthread_mutex_unlock( &cscPortRdy_mutex);
313  2          pthread_exit( NULL );
314  1      }
315  1
316  1      lrc = ElinkGetConnectHandle( ElinkHandle,
317  2                                  ShellHandle,
318  2                                  svc_rpc_h );
319  1
320  1      if ( 0 != lrc )
321  1      {
322  2          EDMDispatch_logent(
323  2              __FILE__, __LINE__, LOG_ERR, DDP_GET_CONNECT_HANDLE_FAILURE,
324  2              0,"ElinkGetConnectHandle() failure");
325  2          UnlockSvcMutex();
326  2          p_so -> setStatus(DD_SERVICE_FAILURE,NONEXEC);
327  2          pthread_exit( NULL );
328  2      }
329  1
330  1      }
331  1
331  1      p_so -> setConnectionHandle((void *)svc_rpc_h);    // Gec csc handle
333  1
334  1      p_so -> getSessionID(&sID);          // Gec Unique Session id
335  1
336  1      // Issue message telling of Dispatch Daemon RDR port number.
337  1
```

```cpp
338  2      if ( isDebugOn() )
339  2      {
340  2          EDMDispatch_logent( __FILE__, __LINE__, LOG_INFO, DDP_PORT_NUMBERS,
341  2              0,"PORT_INFO Dispatch=Daemon,ifspec.portnum");
342  2          DispatchDaemon,ifspec.portnum);
343  2      }
344  1
345  1
346  1      // Unlock Port Rdy mutex so the Reader can listen.
347  1
348  1      pthread_mutex_unlock( &cscPortRdy_mutex);
349  1
350  1      // Tell the Dispatch Daemon Protocol Reader Thread of the
351  1      // restore svc of dispatch protocol details ( port etc ...)
352  1
353  1      pthread_cond_signal(&cscPortRdy_cv);
354  1
355  1      // Send the CCW service handle so we can respond to messages.
356  1
357  1      lrc = edmrst_send_chnd_to_private_svc(fd1);
358  1      if ( 0 != lrc )
359  1      {
360  2          EDMDispatch_logent(
361  2              __FILE__, __LINE__, LOG_ERR, DDP_CHANNEL_SEND_FAILURE,
362  2              0,"edmrst_send_chnd_to_private_svc() failure");
363  2          p_so -> setStatus(DD_SERVICE_FAILURE,NONEXEC);
364  1          UnlockSvcMutex();
365  1          pthread_exit( NULL );
366  1      }
367  1
368  1      // Send the Unique Session id value.
369  1
370  1      lrc = edmrst_send_uid_to_private_svc(fd1,p_so);
371  1      if ( 0 != lrc )
372  1      {
373  2          (void) free(svc_rpc_h);
374  2          EDMDispatch_logent(
375  2              __FILE__, __LINE__, LOG_ERR, DDP_SEND_UID_FAILURE,
376  2              0,"edmrst_send_uid_to_private_svc() failure");
377  2          p_so -> setStatus(DD_SERVICE_FAILURE,NONEXEC);
378  1          UnlockSvcMutex();
379  1          pthread_exit( NULL );
380  1      }
381  1
382  2      lrc = edmrst_create_ddp_client_connection(fd1, &bh, p_so);
383  2      if ( 0 != lrc )
384  2      {
385  2          (void) free(svc_rpc_h);
386  2          EDMDispatch_logent(
387  2              __FILE__, __LINE__, LOG_ERR, DDP_CREATE_CLIENT_CONNECTION,
388  2              0,"edmrst_create_ddp_client_connection() failure");
389  2          p_so -> setStatus(DD_SERVICE_FAILURE,NONEXEC);
390  2          UnlockSvcMutex();
391  1          pthread_exit( NULL );
392  2      }
393  2
394  2  }
395  2
```

```
 396  1      }
 398  1
 399  1      //
 400  1      // Insert handle object into Global list.
 401  1      //
 402  1      lrc = newHandleset( &sID,
 403  1                          fd1,
 404  1                          fd2,
 405  1                          bh,
 406  1                          &bShellHandle,
 407  1                          &status );
 408  1
 409  1      if ( 0 != lrc )
 410  1      {
 411  2         (void) free(svc_rpc_h);
 412  2         EDMDispatch_logent(
 413  2            _FILE_, _LINE_, LOG_ERR, DDP_HANDLE_INSERTION_ERROR,
 414  2            status, "newHandleset() failure");
 415  1         UnlockSvcMutex();
                pthread_exit( NULL );
             }

             //
             // Let's clean up and set the status to RUNNING.
             //
             p.so -> setStatus(DD_SERVICE_RUNNING);
 420  2      UnlockSvcMutex();
 421  1      pthread_exit( NULL );
 422  1      return( NULL );
 423  1
 424  1   }
```

```
 426       /*
 427       ** ==========================================================
 428       **
 429       ** Function:       edrmst_send_chndl_to_private_svc()
 430       **
 431       ** Description:
 432       **
 433       **
 434       **    Returns:        0 Successful
 435       **                   -1 Read Failure
 436       **                   -0 Read less than expected
 437       **
 438       ** ==========================================================
 439       */
 440       int
 441       edrmst_send_chndl_to_private_svc(int pipetoSvc)
 442       {
 444  1       auto int lrc=0;
 445  1       auto unsigned char *p_client_h=NULL;
 446
 447  1       //
 448  1       // Isolate the connection handle from the server 'if_spec',
      1       // The IP/PORT are part of the created if_spec structure
      1       //
      1       p_client_h = Dispatchbaemon_ifspec.connect_handle_p;
 451  1       //
 452  1       // Write the handle to the service so it can contact me
 453  1       //
 454  1       lrc = edrmst_WrChannel(pipetoSvc,
 455  1                              p_client_h,
 456  2                              CONNECT_HANDLE_SIZE);
 457  1
 458  1       if ( CONNECT_HANDLE_SIZE != lrc )
 459  2       {
 460  2          EDMDispatch_logent( _FILE_, _LINE_, LOG_ERR, DDP_WRITE_CHANNEL,
 461  2             0, "edrmst_WrChannel() failure");
 462  1       }
 465  1       return(0);
 466       }
```

```
467   /*
468   **  ================================================================
469   **
470   **  Function:    edmrst_send_uid_to_private_svc()
471   **
472   **  Description:
473   **
474   **
475   **  Returns:     0  Sucessful
476   **              -1  Read Failure
477   **              <0  Read less than expected
478   **
479   **
480   **  ================================================================
481   */
482   int
483   edmrst_send_uid_to_private_svc(int pipersvc,
484                                  EDMSession   *pSessionObj)
485   {
486      auto int lrc=0;
487      auto DD_client_session_id uid;
488
489      //
490      // Write the handle to the service so it can contact me
491      //
492      pSessionObj -> getSessionID(&uid);
493      lrc = edmrst_WrChannel (pipersvc,
494                              (void*)&uid,
495                              sizeof(DD_client_session_id));
496      if ( (sizeof(DD_client_session_id) != lrc )
497      {
498         EXMDispatch_logent( __FILE__, __LINE__, LOG_ERR, DDP_WRITE_CHANNEL,
499                             0, "edmrst_WrChannel() Failure");
500         return(-1);
501      }
502      return(0);
503   }
```

```
502   /*
503   **  ================================================================
504   **
505   **  Function:    edmrst_create_ddp_client_connection()
506   **
507   **  Description:
508   **
509   **
510   **  Returns:     0  Sucessful
511   **              -1  Read Failure
512   **              <0  Read less than expected
513   **
514   **
515   **  ================================================================
516   */
517   int
518   edmrst_create_ddp_client_connection(int pipersvc,
519                                       rpc_binding_handle_t **bbh,
520                                       EDMSession    *p_so )
521   {
522      int lrc;
523      unsigned char *p_restore_service=NULL;
524      error_status_t status;
525      rpc_if_handle_t *p_psvc_ifspec=NULL;
526      rpc_binding_handle_t *psvc_h=NULL;
527
528      //
529      // We now need to get the details from the restore service on
530      // how to connect to the dispatch daemon ccw to the restore
531      // service ccr. At this point, the restore service will be send
532      // the the service ccr handle information. The port / ip
533      // are the key information needed to create the ddp ccw handle
534      //
535      lrc = edmrst_get_client_handle( pipersvc, &p_restore_service );
536      if ( 0 != lrc )
537      {
538         EXMDispatch_logent( __FILE__, __LINE__, LOG_ERR, DDP_GET_CLIENT_HANDLE,
539                             0, "edmrst_get_client_handle() Failure");
540         return(-1);
541      }
542
543      //
544      // Create an ifspec from the handle
545      //
546      p_psvc_ifspec = (rpc_if_handle_t *)
547                      calloc(1,sizeof(rpc_if_handle_t));
548      if (p_psvc_ifspec == NULL)
549      {
550         EXMDispatch_logent( __FILE__, __LINE__, LOG_ERR, DDP_NO_MEMORY,
551                             0, "calloc() failure");
552         return(-1);
553      }
554
555      lrc = csc_private_ifspec_init( p_restore_service,
556                                     EDM_DISPATCH_PROTOCOL_CLIENT,
557                                     EDMDPC_FUNCTIONS,
558                                     p_psvc_ifspec,
559                                     &status );
560      if ( 1 != lrc )
561      {
562         EXMDispatch_logent( __FILE__, __LINE__, LOG_ERR, DDP_IFSPEC_INIT_FAILURE,
563                             status, "csc_private_ifspec_init() Failure");
564         (void) free(p_psvc_ifspec);
565         return(-1);
566      }
```

```
564 1  }
565 2
566 1      if ( isDebugOn() )
567 1      {
568 2          EDMDispatch_logent ( __FILE__, __LINE__, LOG_INFO_DDP_PORT_NUMBERS,
569 2              0, "PORT_INFO p.psvc_ifspec(DDCOM) port# %d',
570 1              p_psvc_ifspec->portnum) ;
571 1      }
572
573 1      psvc_h = (rpc_binding_handle_t *) calloc(1, sizeof(
                   rpc_binding_handle_t));
574 1
575 1      // Using the connect handle (128 bytes) received from the restore
576 1      // service, connect to the async rpc_ddp restore service.
577 1      //
579 1      lrc = csc_connect_to_async_rpc_service (
580 1              NULL,
                   "p_psvc_ifspec,
                   psvc_h,
                   &status );
582 1
583 1      if ( 1 != lrc )
584 2      {
585 1          EDMDispatch_logent(
596 2              __FILE__, __LINE__, LOG_ERR_DDP_PRIVATE_SVC_CONNECT_FAILURE,
587 2              status, "csc_connect_to_async_rpc_service (
                       ) Failure. Status is %d', status);
588 1
588 2          return(-1);
589 1      }
590 1
591 2      (void) free(p_psvc_ifspec);
592 1      (void) free(psvc_h);
594 1      return(0);
595  }
```

```
597
598
599  /*
600   **************************************************
       *
       *  Function:
       *  Description:
       *
       *  Returns:
       *      0 Successful
       *     -1 Read Failure
       *
       **************************************************
616   */
617 1  int
618 1  EDMDDSvcInit()
619 1  {
621 1      struct hostent      *hp;
620 1      struct utsname      name;
612 1      error_status_t      csc_status;
613 1      int lrc = 0;
614 1
618 1      ElinkHandle = ElinkInitAPI(ELINK_SHELL, EDMLINK);
619 1
620 2      if ( ElinkHandle == NULL )
621 1      {
622 1          return -1;
623 1      }
624 1
625 1      // Initialize the ifspec specification from the private svc
626 1      // creation call. This call will output the DispatchDaemon.ifspec
627 1      //
628 2      lrc = csc_async_ifspec_init (&DispatchDaemon.ifspec,
629 2              CSC_IFSPEC_PRIVATE_TYPE,
630 1              DP_PROGNUM,
631 1              DP_VERSNUM,
                   dispatch_func_p_t)&edm_dispatch_protocol_service_1_table,
                   &csc_status);
632 1
635 1      if ( TRUE != lrc )
636 2      {
637 2          EDMDispatch_logent (
638 2              __FILE__, __LINE__, LOG_ERR_DDP_IFSPEC_INIT_FAILURE,
639 1              csc_status, "csc_async_ifspec_init() failure");
640 1
640 1          return(-1);
641 1      }
642 1
643 1      //
644 1      // We need the system name and ip for the if_spec.
645 1      uname( &name );
646 1      hp = gethostbyname( name.nodename );
648 1      if ( NULL == hp )
648 1      {
649 1          EDMDispatch_logent(
650 2              __FILE__, __LINE__, LOG_ERR_DDP_GETHOSTNAME_FAILURE,
651 1              0, "gethostbyname() failure");
651 1
652 1          return -1;
653 1      }
655 1      ( void ) memcpy( (char*) &DispatchDaemon.ifspec.ip_addr,
656 1              hp->h_addr, hp->h_length );
```

```
658  1   //
659  1   // Register the callback functions.
660  1   //
661  1   lrc = csc_register_async_server_interface(
661  1                        &Dispatchdaemon_ifspec,
661  1                        -1,
664  1                        edm_dispatch_protocol_service_l_table,
665  1                        edm_dispatch_protocol_service_l_nproc,
666  1                        &csc_status );

668  1   if ( TRUE != lrc )
668  2   {
668  1       EDMDispatch_logevent(
671  2           __FILE__, __LINE__, LOG_ERR, DDP_REGISTER_SVR_FAILURE,
670  2           csc_status,
                "Failed to register asynchronous server interface." );
672  2       return -1;
673  1   }

675  1       return 0;
676  1   }
```

```
1   /*
2   **
3   ** Copyright 1996, 1997 BMC Corporation
4   **
5   ** ===========================================
6   **
7   ** EDMOD_ccw.c
8   **
9   **
10  ** Mission Statement: This is the entry point for the Control Channel
11  **                    Writer thread.
12  **                    Its main purpose is to write notifications of the
13  **                    progress to the
14  **                    Dispatch Daemon.
15  **
16  ** Primary Data Acted On:
17  **
18  ** Compile-Time Options:
19  **
20  **        USE_SUNRPC - Compile sources with surrpc
21  **                     support. If
22  **                     not sum, assume DCE surrpc support.
23  **
24  ** Basic idea here: Module for Control Channel Writer thread.
25  **
26  ** ===========================================
27  **
28  ** The following provides an RCS id in the binary that can be located
29  ** with the what(1) utility. The intent is to keep this short.
30  **
31  */
32
33  #if !defined(lint)
34  static char   RCS_id [] = "@(#)$RCSfile: EDMODccw.c,v $ "
35                            "$Revision: 1.23 $ "
36                            "$State: 1997/02/06 20:49:15 $ ";
37  #endif
38
39  /* _POSIX_SOURCE     unable to compile with this define set */
40  /* #define _POSIX_SOURCE     unable to keep this define set */
41  /* #define _XOPEN_SOURCE     unable to compile with this define set */
42
43  #include <sel/c_portable.h>
44  #include <sel/ep_xopen.h>
45  #include <sec/sscomm.h>
46
47  #include <cw/collect.h>
48
49  #include <sel/inout.h>
50
51  #ifdef __cplusplus
52  extern "C" {
53  #endif
54
55  #include <restore/dispatch_protocol.h>
56  #include <restore/dispatch_protocol_client.h>
57  #include <restore/sec_dispatch_Protocol_client.h>
```

```
58  #endif
59
60  #include <EDMOD.h>
61  #include <EDMOD_ccp.h>
62  #include <EDMOReturnMessage.h>
63  #include <EMccw.h>
64  #include <EMcilla.h>
65  #include <EDMODHandleMgr.h>
66  #include <EDMODHandleMgrApi.h>
67  #include <EDMODispatchSession.h>
68  #include <logging/logging.h>
69  #include <EDMDispatchlog.h>
70
71  extern ELinkHandlePtr_ty   ELinkHandle;
72
73  /* Internal Routines */
74  static int   SendConnectConfirmmessage()
75
76  static int   SendAbortRequestMessage(
77              DD_client_session_id*.rpc.binding_handle_t*);
78
79  static int   SendCloseMessage(
80              DD_client_session_id*.rpc.binding_handle_t*);
81
82  static int   SendPingRequestMessage(
83              DD_client_session_id*.rpc.binding_handle_t*);
84
85  static int   SendFinalStatusConfirmMessage(
86              DD_client_session_id*.rpc.binding_handle_t*);
87
88  void *
89  DispDaemon_ccw(void *buf)
90  {
91      int   rc;
92      int   rc=0;
93      int   status=0;
94      ResponseMessage   ResponseMessage;
95      int   status=0;
96      DD_client_session_id   sid;
97      rpc_binding_handle_t  *client_h.p=NULL;
98
99      for ( ; ; )
100     {
101         /* Monitor the event queue for messages to send. */
102         rc = PopResponseMessage(&ResponseMessage,
103                                 &client_h.p,
104                                 &status);
105
106         if ( -1 == rc )
107         {
108             sleep( DISPATCH_CCW_SLEEP );
109             continue;
110         }
111
112         rc = GetSessionStatus(sid, &status, &status);
113
114         if ((rc != 0) && (ResponseMessage == dp_ping_request))
115         {
116             (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR,
                      DDP_GET_SESSION_STATUS_FAILURE, status,
                      "Can't retrieve status for session "
                      "%ld!%ld>   -  drop message.",
                      sid.high, sid.low);

                continue;
            }

            if (status == DD_SERVICE_FAILURE_NOEXEC || status ==
                 DD_SERVICE_FAILURE_EXEC ||
                status == DD_SERVICE_FAILURE_PERMS)
            {
                sstatus = DD_SERVICE_FAILURE;

                (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_INFO,
```

```
                              DDP_DROP_MESSAGE, 0,
                         "Session <sid.hid> failed to start - drop
                              message.",
                         sid.high, sid.low);

                     continue;
                 }

         /* execute the callback that will process this message */
         switch( ResponseMessage )
         {
             case dp_connect_confirm:
                 rc = SendConnectConfirmMessage(
                     sid, client_b_p);
                 break;

             case dp_abort_request:
                 rc = SendAbortRequestMessage(
                     &sid, client_b_p);
                 break;

             case dp_close_request:
                 rc = SendCloseRequestMessage(
                     &sid, client_b_p);
                 break;

             case dp_event_confirm:
                 // No confirm needed for this message
                 break;

             case dp_progress_confirm:
                 // No confirm needed for this message
                 break;

             case dp_ping_request:
                 rc = SendPingRequestMessage(&sid, client_b_p);
                 break;

             case dp_final_stats_confirm:
                 rc = SendFinalStatsConfirmMessage(
                     &sid, client_b_p);
                 break;

             default:
                 EDMDispatch_logent(
                     __FILE__, __LINE__, LOG_ERR, DDP_INVALID_MESSAGE,
                     0, "Invalid message type received.");
                 break;
         }

         if (rc != 0)
         {
             sleep(1);
         }

         /* Check for a shutdown setting */
     } /* End of forever loop */
     return((void*)0);
 } /* End of DispDaemon_ccw() */
```

```
//
// Function: SendConnectConfirmMessage()
//
// Description:
//     Send the confirm connect message to the
//     dispatch daemon.
//
static int
SendConnectConfirmMessage(
     DP_client_session_id *ssid, rpc_binding_handle_t *clnt_p)
{
     int *rc = NULL;
     int lrc = 0;
     int savedirc = 0;
     int status = 0;
     int status = 0;
     DP_connect_confirm_msg *msg_p=NULL;

     if (clnt_p != NULL)
     {
         msg_p = (DP_connect_confirm_msg*)
             calloc(1, sizeof(DP_connect_confirm_msg));
         msg_p->sid.high = ssid->high;
         msg_p->sid.low = ssid->low;

         rc = dp_connect_confirm_1(msg_p, *clnt_p);

         if (isdebug0n())
         {
             (void) EDMDispatch_logent(__FILE__, __LINE__, LOG_INFO,
                 DDP_SENDING_MESSAGE, 0,
                 "Sending dp_connect_confirm_1 message.");
         }

         free( msg_p );
     }
     else
     {
         rpc_binding_handle_t *client_handle_p = NULL;
         lrc = GetCSHandle(&sid, &client_handle_p,
             &status);

         if (0 != lrc)
         {
             EDMDispatch_logent(
                 __FILE__, __LINE__, LOG_ERR, DDP_GET_CSC_HANDLE_FAILURE,
                 status,
                 "GetCSHandle failed.");
             savedirc = lrc;
         }

         /* Push message to send onto the queue */
         lrc = PushResponseMessage((int) dp_connect_confirm,
             &sid,
             client_handle_p,
             &status);

         if (0 != lrc)
         {
             EDMDispatch_logent(
                 __FILE__, __LINE__, LOG_ERR, DDP_PUT_RESPONSE_FAILURE, status,
                 "PushResponseMessage failed.");
             savedirc = lrc;
         }
     }
```

```
221  2
223  2           lrc = saveirc;
224  1           return lrc;
226  1        }
227        return(0);
           }
```

```
229
230     //
231     //  Function:  SendAbortRequestMessage()
232     //  Description:
233     //    Send a abort request to a restore service.
234     static int
235     SendAbortRequestMessage(
           DP_client_session_id *ssid, rpc_binding_handle_t *clnt_p )
236  1  {
237  1     int *rc;
238  1     DP_abort_request_msg *msg_p=NULL;
240  1     msg_p = (DP_abort_request_msg*)
241  1       calloc(1, sizeof(DP_abort_request_msg));
242  1     msg_p->ssid.high = ssid->high;
243  1     msg_p->ssid.low  = ssid->low;
244  1     rc = dp_abort_request_1(msg_p, *clnt_p);
246  1     if (isDebugOn())
247  2     {
248  2        (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_INFO,
249  2           DDP_SENDING_MESSAGE, 0,
250  2           "sending dp_abort_request_1 message.");
251  1     }
253  1     free( msg_p );
254  1     return(0);
255     }
```

```
257   //
258   // Function: SendCloseRequestMessage()
259   // Description:
260   //     Send a close request to a restore service.
261   //
262   static int
263   SendCloseRequestMessage(
264           DP_client_session_id *ssid, rpc_binding_handle_t *clnt_p )
265   {
266       int *rc;
267       DP_close_request_msg *msg_p=NULL;
268       msg_p = (DP_close_request_msg*)
269           calloc(1, sizeof(DP_close_request_msg));
270       msg_p->ssid.high = ssid->high;
271       msg_p->ssid.low  = ssid->low;
272       rc = dp_close_request_l(msg_p, *clnt_p);

274       if (IsDebugOn())
275       {
276           (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_INFO,
277                   DDP_SENDING_MESSAGE, 0,
278                   "Sending dp_close_request_l message.");
279       }

281       free( msg_p );
282       return(0);
283   }
```

```
285   //
286   // Function: SendPingRequestMessage()
287   // Description:
288   //     Send a ping request to a restore service.
289   //
290   static int
291   SendPingRequestMessage(
292           DP_client_session_id *ssid, rpc_binding_handle_t *clnt_p )
293   {
294       int *rc;
295       DP_ping_request_msg *msg_p=NULL;
296       msg_p = (DP_ping_request_msg*)
297           calloc(1, sizeof(DP_ping_request_msg));
298       msg_p->ssid.high = ssid->high;
299       msg_p->ssid.low  = ssid->low;
300       rc = dp_ping_request_l(msg_p, *clnt_p);

302       if (IsDebugOn())
303       {
304           (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_INFO,
305                   DDP_SENDING_MESSAGE, 0,
306                   "Sending dp_ping_request_l message.");
307       }

309       free( msg_p );
310       return(0);
311   }
```

```
313
314    //
315    //    Function:  SendFinalStatsConfirmMessage()
316    //    Description:
317    //        Send a ping request to a restore service.
318    //
319    static int
320  1 SendFinalStatsConfirmMessage(
320  1     DP_client_session_id *ssid, rpc_binding_handle_t *clnt_p )
321  1 {
322  1     int status, *rc;
323  1     int out = 1, err = -1;
324  1     int ret = 0;
325  1     rpc_binding_handle_t *ptr;
326  1     DP_final_stats_confirm_msg *msg_p=NULL;
327  1
327  1     if (clnt_p != NULL)
328  2     {
329  2         msg_p = (DP_final_stats_confirm_msg*)
330  3             calloc(1, sizeof(DP_final_stats_confirm_msg));
331  2         msg_p->ssid.high = ssid->high;
332  2         msg_p->ssid.low = ssid->low;
333  2         rc = dp_final_stats_confirm_1(msg_p,*clnt_p);
333  2
335  3         if (isDebugOn())
335  3         {
336  3             (void) EOMDispatch_logent( __FILE__, __LINE__, LOG_INFO,
337  3                 DDP_SENDING_MESSAGE, 0,
338  3                 "Sending dp_final_stats_confirm_1 "
339  3                 message.");
339  3         }
340  2
340  2         free( msg_p );
342  2     }
343  1
345  1     ret = removeSession(ssid, &status);
346  1     if (ret == -1)
346  1     {
347  2         (void) EOMDispatch_logent( __FILE__, __LINE__, LOG_ERR,
348  2             DDP_REMOVE_SESSION_FAILURE, status,
349  2             "Failure removing session instance from list.
350  2             ");
351  1     }
353  1
353  2     ret = getHandleSet(ssid, &out, &err, &status);
354  2     if (ret == -1)
355  2     {
355  2         (void) EOMDispatch_logent( __FILE__, __LINE__, LOG_ERR,
356  2             DDP_GET_HANDLE_SET_FAILURE, status,
357  2             "Failure getting session handles from list.
358  2             ");
360  1     }
361  1     if (out != -1 && err != -1)
362  2     {
363  2         close(out);
364  2         close(err);
365  1     }
367  1     ret = deleteHandleSet(ssid, &ELinkHandle, &status);
368  1     if (ret == -1)
369  1     {
369  2         (void) EOMDispatch_logent( __FILE__, __LINE__, LOG_ERR,
370  2             DDP_DELETE_HANDLE_SET_FAILURE, status,
371  2             "Failure removing session handles from list.
372  2             ");
```

```
373  1     }
375  1     return(0);
376  1 }
```

```c
1
2    /* =================================================================
3    **
4    ** Copyright 1996, 1997 BMC Corporation                           =
5    **                                                                 =
6    ** ================================================================
7    **                                                                 =
8    ** ================================================================
9    **                                                                 =
10   ** Mission Statement: This is the entry point for the Control Channel
11   **            thread. Its main purpose is to read asynchronous        Reader
12   **            messages from the Dispatch Daemon.
13   **
14   ** ================================================================
15   ** Primary Data Acted On:
16   **
17   ** ================================================================
18   ** Basic idea here: Module for Control Channel Reader thread.
19   **
20   ** ================================================================
21   ** Compile-Time Options:
22   **
23   **     USE_SUNRPC  - Compile source with source
24   **                    support.  If
25   **                    not set, assume DCE support.
26   **
27   ** The following provides an RCS id in the binary that can be located
28   ** with the what(1) utility.  The intent is to keep this short.
29   */
30   #if !defined(lint)
31   static char      RCS_id () = {
32
33   "@(#)$RCSfile: EDMCcr.c,v $ "
34   "$Revision: 1.23 $ "
35   "$Date: 1997/02/06 20:49:15 $" ;
36   #endif
37
38   ** #define _POSIX_SOURCE
39   ** #define _XOPEN_SOURCE          unable to compile with this define set */
40                                    unable to compile with this define set */
41   #include <signal.h>
42   #include <esl/c_portable.h>
43   #include <EDMOD_ddp.h>
44   #include <esl/esl_xopen.h>
45   #include <esl/rout.h>
46
47   #include <pthread.h>
48   #include <logging/logging.h>
49   #include <EDMOD_ccr.h>
50   #include <esc/cscomm.h>
51   #include <EDMdispatchlog.h>
52
53   static void halt_service(int);
54   static boolean12 print_error = TRUE;
55
56   extern pthread_cond_t  cscPortRdy_cv;
57   extern pthread_mutex_t cscPortRdy_mutex;
58   void *
```

```c
59   DispDaemon_ccr(void *buf)
60   {
61       int     irc=0;
62       error_status_t      status;
63       rpc_if_handle_t     if_spec;
64       struct esl_timeval  timeout = {5,0};
65
66       // Wait for transient thread to tell me there is something to listen on.
67
68       pthread_mutex_lock( &cscPortRdy_mutex );
69       pthread_cond_wait( &cscPortRdy_cv, &cscPortRdy_mutex );
70       pthread_mutex_unlock( &cscPortRdy_mutex );
71
72   /*
73   **     Let begin to listen for requests.
74   */
75
76       for(;;)
77       {
78           lrc = csc_async_server_listen( (esl_timeval*)&timeout, &status);
79           if ( 1 == lrc )
80
81
82                   0, "Bad returned from listen.");
83
84                   0, LOG_INFO,DDP_FAILED_LISTEN,
85           if ( 1 == lrc )
86
87           if ( 1 == lrc )
88
89               EDMdispatch_logent(
90                   _FILE_, _LINE_, LOG_INFO, DDP_LISTEN_TIMEOUT,
91                   0, "listen() timeout.");
92
93       } /* End of while loop */
94       }
95
96   /*
97   **     Unregister our service upon exit request.
98   */
99       lrc = csc_unregister_async_server_interface(&if_spec, &status);
100          EDMdispatch_logent( _FILE_, _LINE_, LOG_INFO,DDP_UNREGISTER_SVC,
101              0, "Returned from unregister service.");
102      return((void*)0);
```

```
/*
 *
 *
 *  Copyright 1996,1997 EMC Corporation
 *
 */
/*  EMCDDHandle.cc
 *
 *  Mission Statement:  file that contains the Handle class methods
 *
 *  Primary Data Acted On:
 *
 *  Compile-Time Options:
 *
 *  Basic idea here:
 *
 *      The Handle object is a container which holds a
 *      set of handles for each running service.
 */

#if !defined(lint)
static char    RCS_id [] = "@(#)$RCSfile: EMCDDHandle.cc,v $ "
                           "$Revision: 1.0 $ "
                           "$Date: 1997/02/06 20:49:15 $";
#endif

#include <esl/rc_portable.h>
#include <esl/esp_xopen.h>
#include <esl/inout.h>

#include <stdio.h>
#include <stdlib.h>

// Rogue Wave includes
#include <string.h>
#include <rw/vstream.h>
#include <rw/rwfile.h>
#include <rw/collect.h>

#include <csr/cscomm.h>

#include <restore/ObjectID.h>
#include <restore/RestoreObjectID.h>
#include <restore/dispatch_daemon.h>
#include <EMCDDHandle.h>
#include <edmlink/edmlink_api.h>

// Needed for rogue wave linked list manager.
// d13 is the object ID.
RWDEFINE_COLLECTABLE(EMCDDHandle, EMCDDHANDLE)

/********************************************************************
 **
 **  Routine:    EMCDDHandle constructor
 **
 **  Inputs:     None
 **
 **  Outputs:    None
 **
 **  Return Codes:
 **      None
 **
 **  Purpose:    Initializes the Handle class by resetting the internal
 **              data.
 **
 ********************************************************************
```

```
 */
EMCDDHandle::EMCDDHandle()
{
    rpcBD      = NULL;
    sesslid    = NULL;
    stdoutpipe = 0;
    stderrpipe = 0;

    memset(&sessionID, 0, sizeof(sessionID));
}

/********************************************************************
 **
 **  Routine:    EMCDDHandle constructor
 **
 **  Inputs:     rpc_binding_handle_t *bh - the client handle to use for the
 **                                         dispatch protocol
 **              int stdoutpipe - the stdout descriptor of the service
 **              int stderrpipe - the stderr descriptor of the service
 **              DD_client_session_id sess - the session ID of the
 **                                          service
 **
 **  Outputs:    None
 **
 **  Return Codes:
 **      None
 **
 **  Purpose:    Initializes the internals of the Handle class.
 **
 ********************************************************************
 */
EMCDDHandle::EMCDDHandle(
                IN rpc_binding_handle_t *bh, IN DD_client_session_id *sess,
                IN int stdoutpipe, IN int stderrpipe)
{
    rpcBD = bh;
    stdoutpipe = stdoutpipe;
    stderrpipe = stderrpipe;

    if (sess != NULL)
        memcpy(&sessionID, sess, sizeof(DD_client_session_id));
}

/********************************************************************
 **
 **  Routine:    EMCDDHandle destructor
 **
 **  Inputs:     None
 **
 **  Outputs:    None
 **
 **  Return Codes:
 **      None
 **
 **  Purpose:    Doesn't really do anything but seems to be a requirement
 **              for the linked list manager.
 **
 ********************************************************************
 */
EMCDDHandle::~EMCDDHandle()
{
}
```

```
123  {
124  }
126

     /*******************************************************************
     **
     ** Routine:    compareTo
     **
     ** Inputs:     RWCollectable *c - a pointer to the base class type which
     **                                you can then cast and compare.
     **
     ** Outputs:    None
     **
     ** Return Codes:
     **             int     -     returns numbers like qsort compare (-1, 0, 1)
     **
     ** Purpose:    Compare using the auxproc PID.
     **
     *******************************************************************/

     int
     EDMDHandle::compareTo(IN const RWCollectable *c) const
     {
         EDMDHandle *localhandle = (EDMDHandle *) c;

         if (localhandle == NULL)
             return -1;

         if (localhandle -> sessionID.high == sessionID.high &&
             localhandle -> sessionID.low == sessionID.low)
             return 0;

         return (localhandle -> sessionID.high > sessionID.high ||
                (localhandle -> sessionID.high == sessionID.high &&
                 localhandle -> sessionID.low > sessionID.low)) ? 1 : -1;
     }

     /*******************************************************************
     **
     ** Routine:    isEqual
     **
     ** Inputs:     RWCollectable *c - a pointer to the base class type which
     **                                you can then cast and compare.
     **
     ** Outputs:    None
     **
     ** Return Codes:
     **             RWBoolean   -    TRUE or FALSE
     **
     ** Purpose:    Compare session IDs to find which session needs service.
     **
     *******************************************************************/

     RWBoolean
     EDMDHandle::isEqual(IN const RWCollectable *c) const
     {
         EDMDHandle *localhandle = (EDMDHandle *) c;

         if (localhandle == NULL)
             return FALSE;

         if (localhandle -> sessionID.high == sessionID.high &&
```

```
             localhandle -> sessionID.low == sessionID.low)
             return TRUE;
         else
             return FALSE;
     }

     /*
     **
     ** Routine:    hash
     **
     ** Inputs:     None
     **
     ** Outputs:    None
     **
     ** Return Codes:
     **             unsigned    -    returns time unique value, in this case started
     **
     ** Purpose:    Returns unique value, in this case auxproc pid.
     **
     */

     unsigned
     EDMDHandle::hash() const
     {
         return (unsigned) sessionID.low;
     }

     /*
     **
     ** Routine:    saveGuts
     **
     ** Inputs:     RWFile   f  -  File pointer where data will be saved.
     **
     ** Outputs:    None
     **
     ** Return Codes:
     **             None
     **
     ** Purpose:    Save class internal data to a file.
     **
     */

     void
     EDMDHandle::saveGuts(IN RWFile &f)
     {
         // Save parent class data too
         RWCollectable::saveGuts(f);

         // left as an example

     /*
     **
     ** Routine:    saveGuts
     **
     ** Inputs:     RWostream strm - stream to write internal data to.
     **
     ** Outputs:    None
     **
```

```
246    **  Return Codes:
247    **      None
248    **
249    ** */
250    void
251    EDMDDHandle::saveGuts(IN RWvostream &strm)
252    {
           // Save parent class data too
254        RWCollectable::saveGuts(strm);
255 1
256 1
257 1      // Left as an example
258 1  }

263    /*******************************************************
264    **
       **  Routine:    restoreGuts
       **
       **  Purpose:    Restores an instance of the Handle class by reading
                       the data
       **              from the passed in file.
       **
       **  Inputs:     RWFile f - file to read internal data from.
       **
       **  Outputs:    None
       **
       **  Return Codes:
       **      None
       **
       ** */
```

```
278    /*******************************************************
       **
280    **  Routine:    restoreGuts
281    **
282    **  Purpose:    Restores an instance of the Handle class by reading
                       the
       **              data from the passed in stream.
       **
       **  Inputs:     RWvistream strm - stream to read internal data from.
283 1  **
284 1  **  Outputs:    None
       **
286    **  Return Codes:
287        **      None
       **
289    ** */
       void
290    EDMDDHandle::restoreGuts(IN RWFile &f)
291 {
292 1      // Restore parent class data too
293        RWCollectable::restoreGuts(f);
294 1
295 1      // Left as an example
296 1      None
297 1  }
298 1
299 1
300
301    /*******************************************************
302    **
303    **  Purpose:    Restores parent data too
304
305    */
```

```
306    void
307    EDMDDHandle::restoreGuts(IN RWvistream &strm)
308 {
309 1      // Restore parent class data too
310 1      RWCollectable::restoreGuts(strm);
311 1
312 1      // Left as an example
313 1  }
315    /*******************************************************
316    **
317    **  Routine:    binaryStoreSize
318    **
319    **  Purpose:    Returns the size of class if it were stored on disk.
320    **
321    **  Inputs:     None
322    **
323    **  Outputs:    None
324    **
325    **  Return Codes:
326    **      RWspace count - file size of class written to disk in
                   bytes
327    **
328    ** */
329    RWspace
       EDMDDHandle::binaryStoreSize() const
331 {
       RWspace count = RWCollectable::binaryStoreSize();
333 1      return count;
334 1  }
335 1
336 1
338    /*******************************************************
339    **
340    **  Routine:    getSessionID
341    **
342    **  Purpose:    Returns the ID of the session the object belongs to.
343    **
344    **  Inputs:     None
345    **
346    **  Outputs:    None
347    **
348    **  Return Codes:
349    **      DD_client_session_id sessionID - the session ID
350    **
351    ** */
352    DD_client_session_id
354    EDMDDHandle::getSessionID()
355 {
356 1      return sessionID;
357 1  }
358 1
360    /*******************************************************
361    **
362    **  Routine:    setSessionID
363    **
364    **  Inputs:     DD_client_session_id ID - the session ID associated with
```

```
363 **    Inputs:     ID_client_session_id ID - the ID of the session
                                               this object
365 **
366 **    Outputs:    None
367 **
368 **    Return Codes:
369 **        None
370 **
371 **    Purpose:    Sets the ID of the session the object belongs to.
372 **
373 **
374 **
375 */
377 void
378 EDMDDHandle::setSessionID(ID_client_session_id ID)
379 {
380 1     sessionID = ID;
381 }
383 /********************************************
384 **
385 **    Routine:    getBindingHandle
386 **
387 **    Inputs:     None
388 **
389 **    Outputs:    None
390 **
391 **    Return Codes:
392 **        rpc_binding_handle_t rpcBD - the binding handle for the
                                          client
393 **                                    side of the dispatch protocol
395 **
396 **    Purpose:    Return the binding handle for the client side of the
397 **                dispatch
398 **
399 */
401 rpc_binding_handle_t *
402 EDMDDHandle::getBindingHandle()
403 {
405 1     return rpcBD;
407 }
408
409 /********************************************
410 **
411 **    Routine:    setBindingHandle
411 **    Inputs:     rpc_binding_handle_t *bh - the binding handle to use for
                                                this
412 **                                          service
413 **
414 **    Outputs:    None
415 **
416 **    Return Codes:
417 **        None
418 **
419 **    Purpose:    Sets the binding handle of the session the object belongs
420 **                to.
421 **
```

```
422 */
424 void
425 EDMDDHandle::setBindingHandle(rpc_binding_handle_t *bh)
426 {
427 1     rpcBD = bh;
428 }
430 /********************************************
431 **
432 **    Routine:    getShellHandle
433 **
434 **    Inputs:     None
435 **
436 **    Outputs:    None
437 **
438 **    Return Codes:
439 **        ELinkShellObjPtr_ty - the shell handle for the client
                                   side of the dispatch protocol
440 **
442 **    Purpose:    Returns the shell handle for the client side of the
443 **                protocol.
445 */
448 ELinkShellObjPtr_ty *
449 EDMDDHandle::getShellHandle()
450 {
451 1     return &shellHd;
452 }
454 /********************************************
455 **
456 **    Routine:    setShellHandle
457 **
458 **    Inputs:     ELinkShellObjPtr_ty *bh - the shell handle to use for
                                               this
459 **                                         service
460 **
462 **    Outputs:    None
463 **
464 **    Return Codes:
465 **        None
466 **
467 **    Purpose:    Sets the shell handle of the session the object belongs
468 **                to.
469 */
471 void
472 EDMDDHandle::setShellHandle(ELinkShellObjPtr_ty *bh)
473 {
474 1     shellHd = *bh;
475 }
477 /********************************************
478 **
```

```
479  **    Routine:    getStdoutPipe
480  **
481  **    Inputs:     None
482  **
483  **    Outputs:    None
484  **
485  **    Return Codes:
486  **          int stdoutFD - the stdout descriptor of the service
487  **
488  **    Purpose:    Returns the stdout handle of the service.
489  **
490  **
491  ***********************************************************
492  */
493  int
494  EDMDHandle::getStdoutPipe()
495  {
496      return stdoutPipe;
497  }


499  /***********************************************************
500  **
501  **    Routine:    setStdoutPipe
502  **
503  **    Inputs:     int handle - the stdout handle of the private service
504  **
505  **    Outputs:    None
506  **
507  **    Return Codes:
508  **          None
509  **
510  **    Purpose:    Sets the stdout handle of the private service.
511  **
512  **
513  ***********************************************************
514  */
515  void
516  EDMDHandle::setStdoutPipe(int handle)
517  {
518      stdoutPipe = handle;
519  }


521  /***********************************************************
522  **
523  **    Routine:    getStderrPipe
524  **
525  **    Inputs:     None
526  **
527  **    Outputs:    None
528  **
529  **    Return Codes:
530  **          int stderrPipe - the stderr descriptor of the service
531  **
532  **    Purpose:    Returns the stderr descriptor of the service.
533  **
534  **
535  ***********************************************************
536  */
537  int
538  EDMDHandle::getStderrPipe()
539  {
```

```
540      return stderrPipe;
541  }

543  /***********************************************************
544  **    Routine:    setStderrPipe
545  **
546  **    Inputs:     int handle - the stderr handle of the service
547  **
548  **    Outputs:    None
549  **
550  **    Return Codes:
551  **          None
552  **
553  **    Purpose:    Sets the stderr handle of the service.
554  **
555  **
556  ***********************************************************
557  */
559  void
560  EDMDHandle::setStderrPipe(int handle)
561  {
562      stderrPipe = handle;
563  }
```

```
1   /*
2   ** Copyright 1996,1997 EMC Corporation
3   */
4
5   // EDMDDHandleMgrApi.cc
6   *
7   *
8   *
9   *    Mission Statement: An API to manage the handle sets/objects
10  *
11  *    Primary Data Acted On:
12  *
13  *    Compile-Time Options:
14  *
15  *
16  *    Basic idea here:
17  *
18  *              This API manages the handle sets.
19  *
20  *              need access to the handles to do IO. Multiple threads
21  *
22  #if !defined(lint)
23  *              Each time an fd_set is modified or used we lock
24  *              a mutex to make sure access is serialized.
25  static char   RCS_id [] = "@(#)$RCSfile: EDMDDHandleMgrApi.cc,v $ "
26                            "$Revision: 1.0 $ "
27                            "$Date: 1997/02/08 20:49:15 $ ";
28  #endif
29  #include <EDMDDHandleMgrApi.h>
30  #include <esl/c_portable.h>
31  #include <esl/ep_xopen.h>
32  #include <esl/input.h>
33  #include <stdlib.h>
34  #include <sys/types.h>
35  #include <sys/time.h>
36  #include <pthread.h>
37
38  // The tree that we keep handles in. We could have used any
40  // rogue wave object but I decided to use the tree.
41
42
43  static RWBinaryTree G_handleTree;
45  // These are the fd_sets managed on behalf of the user. The
46  // Modified sets are changed any time new handles are added
47  // and the others are the actual copies given to the user.
48  // This allows the manager to add handles from 1 thread without
49  // directly effecting a select call made in another thread.
51  fd_set  stdoutSetModified,
52          stdoutSet,
53          stderrSetModified,
54          stderrSet;
56  // These are values for the highest number handle that is part of a
57  // given set. Keep in mind that 1 has to be added to this number to
58  // select on the highest handle used.
60  int  highStdout = 0,
61       highStderr = 0;
63  static pthread_mutex_t  G_fdSetMtx = PTHREAD_MUTEX_INITIALIZER;
64  static pthread_mutex_t  G_handleTreeMtx = PTHREAD_MUTEX_INITIALIZER;
```

```
66  /***************************************************
67   *
68   *  Routine:      initFdSets
69   *
70   *  Inputs:       None
71   *
72   *  Outputs:      None
73   *
74   *  Return Codes: None
75   *
76   *
77   *  Purpose:      Initialize fd sets and mutex.
78   *
79   *
80   ***************************************************/
82  static void
83  initFdSets()
84  {
85      FD_ZERO(&stdoutSetModified);
86      FD_ZERO(&stdoutSet);
87      FD_ZERO(&stderrSetModified);
88      FD_ZERO(&stderrSet);
89
90      pthread_mutex_init(&G_fdSetMtx, NULL);
91  }
```

```
 93    /********************************************************
 94    **
 95    **  Routine:    LockHandleMutex
 96    **
 97    **  Inputs:     None
 98    **
 99    **  Outputs:    None
100    **
101    **  Return Codes:
102    **              None
103    **
104    **  Purpose:    Lock the mutex for the handle tree object
105    **
106    **
107    */
108    static void
109    LockHandleMutex()
110    {
111        static boolean_ty first = TRUE;
112
114        if (first == TRUE)
115        {
116            first = FALSE;
117            pthread_mutex_init(&G_handleTreeMtx, NULL);
118        }
119        pthread_mutex_lock(&G_handleTreeMtx);
120    }
121
```

```
123    /********************************************************
124    **
125    **  Routine:    UnlockHandleMutex
126    **
127    **  Inputs:     None
128    **
129    **  Outputs:    None
130    **
131    **  Return Codes:
132    **              None
133    **
134    **  Purpose:    Unlock the mutex for the handle tree object
135    **
136    **
137    */
139    static void
140    UnlockHandleMutex()
141    {
142        pthread_mutex_unlock(&G_handleTreeMtx);
143    }
```

```c
/*****************************************************************
 **
 ** Routine:  getStdoutSet
 **
 ** Inputs:  None
 **
 ** Outputs:  None
 **
 ** Return Codes:
 **       fd_set * - the fd_set...
 **
 ** Purpose:  Returns the stdoutSet fd_set after the most recent copy.
 **       copied into it. Modified is the stdoutSetModified was
 **
 */
int
getStdoutSet(fd_set *yourset, int *highhandle, int *status)
{
    if (status == NULL)
    {
        return -1;
    }

    if (yourset == NULL || highhandle == NULL)
    {
        *status = HANDLEMGR_BAD_PARAM;
        return -1;
    }

    pthread_mutex_lock(&G_fdSetMtx);

    stdoutSet = stdoutSetModified;

    memcpy(yourset, &stdoutSet, sizeof(fd_set));

    *highhandle = highStdout;

    pthread_mutex_unlock(&G_fdSetMtx);

    return 0;
}
```

```c
/*****************************************************************
 **
 ** Routine:  getStderrSet
 **
 ** Inputs:  None
 **
 ** Outputs:  fd_set * - the fd_set for the from descriptors
 **       int    *highhandle - the highest handle for this set
 **       int    *status - there's no status to return right now
 **                but leave it as a placeholder
 **
 ** Return Codes:
 **       int - 0 for success or -1 for failure.
 **
 ** Purpose:  Returns the stderrSet fd_set after the most recent copy.
 **       copied into it. Modified is the stderrSetModified was
 **
 */
int
getStderrSet(fd_set *yourset, int *highhandle, int *status)
{
    if (status == NULL)
    {
        return -1;
    }

    if (yourset == NULL || highhandle == NULL)
    {
        *status = HANDLEMGR_BAD_PARAM;
        return -1;
    }

    pthread_mutex_lock(&G_fdSetMtx);

    stderrSet = stderrSetModified;

    memcpy(yourset, &stderrSet, sizeof(fd_set));

    *highhandle = highStderr;

    pthread_mutex_unlock(&G_fdSetMtx);

    return 0;
}
```

```
236    /******************************************************************
237     **
238     ** Routine:    LookupHandleSet
239     **
240     ** Inputs:     DD_client_session_id *sess - the session ID to lookup
241     **                                            with
242     **             int *status - a place to put a status if something goes
243     **                                            wrong.
244     **
245     ** Outputs:    EDMDDHandle **hs - the handle set to return
246     **
247     ** Return Codes:
248     **             0 for success and non-zero otherwise
249     **
250     ** Purpose:    Looks up a handle set.
251     **
       *******
253    */
254    int
       LookupHandleSet(
255  1     DD_client_session_id *sess, EDMDDHandle **hs, int *status)
256  1 {
258  2     EDMDDHandle *ret, *handleobj;
259  2
260  2     if (status == NULL)
261  1     {
           return -1;
           }
263  1     if (hs == NULL || sess == NULL)
264  2     {
265  2         *status = HANDLEMGR_BAD_PARAM;
266  2         return -1;
267  2     }
269  1     handleobj = new EDMDDHandle();
271  1     if (handleobj == NULL)
272  2     {
273  2         *status = HANDLEMGR_NO_MEMORY;
274  2         return -1;
275  2     }
277  1     handleobj -> setSessionID(*sess);
279  1     LockHandleMutex();
281  1     ret = (EDMDDHandle *) G_handleTree.find(handleobj);
283  1     UnlockHandleMutex();
285  1     delete handleobj;
287  1     if (ret == NULL)
288  2     {
289  2         *status = HANDLEMGR_LOOKUP_FAILED;
290  2         return -1;
291  1     }
293  1     *hs = ret;
295  1     return 0;
296  1 }
```

```
296    }
```

```c
298
299  /******************************************************************
300   **
301   **  Routine:      newHandleSet
302   **
303   **  Inputs:       int stdouthandle - the handle to send commands to auxproc
304   **                int stderrhandle - the handle to receive responses from
305   **                                   auxproc
306   **                rpc_binding_handle_t *commandhandle - the connection handle
307   **                ELinkShellObjPtr_ty *shell - the shell handle
308   **
309   **  Outputs:      int *status - a place to put a status if something goes
310   **                              wrong.
311   **
312   **  Return Codes:   0 for success and non-zero otherwise
313   **
314   **  Purpose:      Creates a new handle set.
315   **
316   **
317   ******************************************************************/
318  int
319  newHandleSet (
320      DD_client_session_id *sess, int stdouthandle, int stderrhandle,
321      rpc_binding_handle_t *commandhandle,
322      ELinkShellObjPtr_ty *shell,
323      int *status)
324  {
325  static boolean_ty first = TRUE;

327  EDMDDHandle *handle;
328  EDMDDHandle *ret;
329  int flags = 0;

331      return -1;
333  if (status == NULL) {

335      return -1;
336  }

338  if (sess == NULL || commandhandle == NULL || shell == NULL) {

342      *status = HANDLEMGR_BAD_PARAM;
343      return -1;
344  }

346  if (first == TRUE) {
347      initFDSets();
348      first = FALSE;
349  }

352  handle = new EDMDDHandle();

354      *status = HANDLEMGR_NO_MEMORY;
355      return -1;

356  handle -> setStdoutPipe(stdouthandle);
     handle -> setStderrPipe(stderrhandle);
     handle -> setBindingHandle(commandhandle);
     handle -> setShellHandle(shell);
     handle -> setSessionID(*sess);
```

```c
358  pthread_mutex_lock(&g_fdSetMtx);

360      flags = fcntl(stdouthandle, F_GETFL, 0);
361      fcntl(stdouthandle, F_SETFL, flags | O_NDELAY);

363      flags = fcntl(stderrhandle, F_GETFL, 0);
364      fcntl(stderrhandle, F_SETFL, flags | O_NDELAY);

366      FD_SET(stdouthandle, &stdoutSetModified);
367      FD_SET(stderrhandle, &stderrSetModified);

369      if (stdouthandle > highStdout)
370          highStdout = stdouthandle;

372      if (stderrhandle > highStderr)
374          highStderr = stderrhandle;

377  pthread_mutex_unlock(&g_fdSetMtx);

379  LockHandleMutex();

381  ret = (EDMDDHandle *) g_handleTree.insert(handle);

383  if (ret == NULL) {

387      *status = HANDLEMGR_INSERT_FAILED;
388      delete handle;
389      return -1;
390  }

392  UnlockHandleMutex();

394      return 0;
395  }
```

```
397    /* ***************************************************************
398     **
399     **
400     ** Routine:      getHandleSet
401     **
402     ** Inputs:       DD_client_session_id *sess - a session ID to use to look
403     **                                            up
404     **                                            the handle set
405     **
406     ** Outputs:      int *status - a place to put a status if something goes
407     **                             wrong.
408     **               int *sout   - stdout descriptor for the service.
409     **               int *serr   - stderr descriptor for the service.
410     **
411     ** Return Codes:
412     **               0 for success and non-zero otherwise
413     **
414     ** Purpose:      Removes a handle set.
415     **
416     **
417     ***************************************************************** */
418    int
419    getHandleSet(
420        IN DD_client_session_id *sess, OUT int *sout, OUT int *serr,
421        OUT int *status)
422    {
423        EDMDHandle *handle;
424        int        lret;
425
426        if (sess == NULL || sout == NULL || serr == NULL)
427        {
428            *status = HANDLER_BAD_PARAM;
429            return -1;
430        }
431
432        if (status == NULL)
433        {
434            return -1;
435        }
436
437        lret = LookupHandleSet(sess, &handle, status);
438
439        if (lret != 0)
440        {
441            return lret;
442        }
443
444        *sout = handle -> getStdoutPipe();
445        *serr = handle -> getStderrPipe();
446
447        return 0;
448    }
```

```
447    /* ***************************************************************
448     **
449     **
450     ** Routine:      GetCSCHandle
451     **
452     ** Inputs:       DD_client_session_id *sess - a session ID to use to look
453     **                                            up
454     **                                            the handle set
455     **
456     ** Outputs:      int *status - a place to put a status if something goes
457     **                             wrong.
458     **               rpc_binding_handle_t *cscb - binding handle for this
459     **                                            session
460     **
461     ** Return Codes:
462     **               0 for success and non-zero otherwise
463     **
464     ** Purpose:      Returns CSC binding handle.
465     **
466     ***************************************************************** */
467    int
468    GetCSCHandle(
469        IN DD_client_session_id *sess, OUT rpc_binding_handle_t **cscb,
470        OUT int *status)
471    {
472        EDMDHandle *handle;
473        int        lret;
474
475        if (sess == NULL || cscb == NULL || status == NULL)
476        {
477            return -1;
478        }
479
480        lret = LookupHandleSet(sess, &handle, status);
481
482        if (lret != 0)
483            return lret;
484        else if (handle == NULL)
485            return -1;
486
487        *cscb = handle -> getBindingHandle();
488
489        return 0;
490    }
```

```
489
490   /*  *************************************************
491   **
492   **  Routine:   GetShellHandle
493   **
494   **  Inputs:    DD_client_session_id *sess  - a session ID to use to look
495                                                up
496   **             the handle set
497   **  Outputs:   int *status  - a place to put a status if something goes
498                                                wrong.
499   **             ElinkShellObjPtr_ty **shell - shell handle for this
500                                                session
501   **
502   **  Return Codes:
503   **             0 for success and non-zero otherwise
504   **
505   **  Purpose:   Returns shell handle.
506   **
507   **  *************************************************
508   */
509   int
510   GetShellHandle(
511       IN DD_client_session_id *sess, OUT ElinkShellObjPtr_ty **shell,
512                                      OUT int *status)
513 1 {
514 1     EDMDDHandle *handle;
515 1     int         lret;
516 2
517 1     if (sess == NULL || shell == NULL || status == NULL)
518 2     {
519 2         return -1;
520 2     }
521 1
522 1     lret = LookupHandleSet(sess, &handle, status);
523 1
524 1     if (lret != 0)
525 2     {
526 2         return lret;
527 2     }
528 1     else if (handle == NULL)
529 2     {
         *shell = handle -> getShellHandle();
         return 0;
      }
}
```

```
531   /*  *************************************************
532   **
533   **  Routine:   deleteHandleSet
534   **
535   **  Inputs:    DD_client_session_id *sess - a session ID to use to look
536                                                up
537   **             the handle set
538   **  Outputs:   int *status - a place to put a status if something goes
539                                                wrong.
540   **
541   **  Return Codes;
542   **             0 for success and non-zero otherwise
543   **
544   **  Purpose:   Removes a handle set.
545   **
546   **  *************************************************
547   */
548   int
549   deleteHandleSet(IN DD_client_session_id *sess, ElinkHandlePtr_ty *hand,
550                   int *status)
551 1 {
552 1     EDMDDHandle *handle;
553 1     int          lret;
554 1     int          ret;
555 1     rpc_binding_handle_t thh;
556 1     ElinkShellObjPtr_ty *shell;
557 1     error_status_t err;
558 1
559 1     if (status == NULL)
560 2     {
561 2         return -1;
562 2     }
563 1
564 1     if (sess == NULL)
565 2     {
566 2         *status = HANDLEMGR_BAD_PARAM;
567 2         return -1;
568 2     }
569 1
570 1     lret = LookupHandleSet(sess, &handle, status);
571 1
572 1     if (lret != 0)
573 2     {
574 2         return lret;
575 2     }
576 1     else if (handle == NULL)
577 2     {
578 2         return -1;
579 2     }
580 1
581 1     LockHandleSetMutex();
582 1
583 1     ret = (EDMDDHandle *) G_handleTree.remove(handle);
584 1
585 1     UnlockHandleSetMutex();
586 1
587 1     if (ret != NULL)
588 2     {
589 2         *status = HANDLEMGR_REMOVE_FAILED;
590 2         delete handle;
591 2         return -1;
592 2     }
593 1
594 1     pthread_mutex_lock(&G_fdSetmtx);
```

```
592  1     FD_CLR(ret -> getStdoutPipe(), &stdoutSetModified);
593  1     FD_CLR(ret -> getStderrPipe(), &stderrSetModified);

595  1     pthread_mutex_unlock(&O_fdSetMtx);

597  1     bn = ret -> getBindingHandle();

599  1     csc_free_binding(bn, O, &err);

601  1     shell = ret -> getShellHandle();

603  1     ELinkDestroyObj((hand, *shell);

605  1     delete ret;

607  1     return 0;
608  1  }
```